Lab Exercise #4: JSON Mapping with Embedded Tomcat

Important! The techniques discussed in this document MUST be used in order for the Group Project to work.

The ‘Rebellion’ Against EE Complexity!

After completing this Lab, you will have achieved the following

1) Learned about alternative run-time environments often used instead of Jakarta EE
2) Create a REST endpoint in less than 100 lines of code (not counting comments!)
3) Just like JPA handles a variety of mappings, JSON Mapping handles familar scenarios

History of EE Complexity

In this course, we have used about a dozen EE components: Servlets, JSPs, Managed Beans, EJBs, Entity Manager as well as **Contexts and Dependency Injection** (CDI) – `@Resource`, `@Inject`, `@PersistenceContext` and `@EJB`. Using EE version 8, the use of annotations have helped keep things simple: the external configuration in ‘deployment descriptors’ XML files (`web.xml`, `persistence.xml`, etc.) is minimal.

However, this was not always the case! In past EE versions – J2EE versions 1.2 through to Java EE 5 (4 major releases/many minor releases, across approx 10 years) – things were ***very*** difficult to configure. Additionally, vendor-specific settings also contributed to complexity. Setting up one’s very first production EE Application required a lot of time and experience, or the help of expensive consultants 😳 !

How did folks deal with EE Complexity?

The ‘EE Way’ to deal with complexity was to reduce the number of components. In Java EE 6, a subset of the ‘Full Profile’ – the ‘Web Profile’ – was created with approximately half the number of components. However, the ‘Web Profile’ was not hugely successful since vendors had no financial incentive to support it: the Software-&-Support fees for ‘Full Profile’ generated ***billions*** in revenue!

Open Source Application Servers

The EE standards took a very long time to mature (process continues today). During that time, a number of Open Source projects created Application Servers with ***some*** of EE capabilities: Apache Tomcat®, Eclipse Jetty, others that were both Open and Commercial such as JBoss (now ‘Wildfly’) or IBM Websphere (now ‘Open Liberty’) These low-cost alternatives cut into the revenue-streams of ‘Full Profile’ Java EE App Servers: not overnight, but slowly over many years. In 2017, after years of declining revenue and interest by its customers, Oracle turned over the majority of its Java EE codebase(s) and Intellectual Property to the Eclipse Consortium.

Tomcat Application Server

Ironically, the Open Source project many folks switched to – and are still using in production today – is Apache Tomcat® Server, whose primary components are Java Servlets and JavaServer Pages– additional components can be easily added.

A Little History …

https://www.oreilly.com/library/view/tomcat-the-definitive/9780596101060/ch01s05.html
Brittain, J. and Darwin, I. (2007). *Tomcat: The Definitive Guide, 2nd Edition*. Champaign, IL (USA): O'Reilly Media, Inc.

A software engineer employed by Sun Microsystems - James Duncan Davidson (who also authored other remarkable Java projects) – rewrote the core of an earlier Servlet implementation. Tomcat therefore became a Reference Implementation for Servlets – at first it was not official, but the software kept passing all the tests while others could not! He then convinced Sun to donate the new Server to the Apache Software Foundation. To this day, a large portion of Payara's internal classes are derived from Tomcat source code (same for Wildfly, Liberty and many others). The use of Tomcat is extremely wide-spread: my Smart TV runs it and I have also seen Smart toasters, dishwashers and fridges use Tomcat !!

Spring Framework

Separate from the folks using Apache Tomcat®, another 'Rebellion Against EE Complexity' came from Rod Johnson in 2002: the **Spring Framework** (https://spring.io) Johnson was a well-known consultant and author who published a book *Expert One-on-One J2EE Design and Development* containing about 30K lines of code (originally called 'interface21') that focused on reducing EE complexity, long before 'Web Profile' existed. Over the next decades-&-a-half, ***thousands*** of paying (!) customers switched away from proprietary EE Servers to use Spring, specifically 'Spring Boot' https://spring.io/projects/spring-boot

Tomcat Embedded

The Apache Tomcat® project has a version of its Server that is ***embedded***: one creates a single Java `main` application and with only about 100 lines of code, REST'ful endpoints can be built. This simplicity and size is remarkable: Tomcat Embedded has a dependency-tree of about 30 components vs. Payara Server which uses over ***500*** components:

```
mvn -DskipTests=true dependency:tree -Dscope=compile
[INFO] +- org.slf4j:slf4j-api:jar:1.7.32:compile
[INFO] +- ch.qos.logback:logback-classic:jar:1.2.9:compile
[INFO] |  \- ch.qos.logback:logback-core:jar:1.2.9:compile
[INFO] +- ch.qos.logback:logback-access:jar:1.2.9:compile
[INFO] +- org.slf4j:jul-to-slf4j:jar:1.7.32:compile
[INFO] +- commons-io:commons-io:jar:2.11.0:compile
[INFO] +- org.glassfish.jersey.core:jersey-client:jar:2.35:compile
[INFO] |  +- jakarta.ws.rs:jakarta.ws.rs-api:jar:2.1.6:compile
[INFO] |  +- org.glassfish.jersey.core:jersey-common:jar:2.35:compile
[INFO] |  |  +- jakarta.annotation:jakarta.annotation-api:jar:1.3.5:compile
[INFO] |  |  \- org.glassfish.hk2:osgi-resource-locator:jar:1.0.3:compile
[INFO] |  \- org.glassfish.hk2.external:jakarta.inject:jar:2.6.1:compile
[INFO] +- org.glassfish.jersey.inject:jersey-hk2:jar:2.35:compile
[INFO] |  +- org.glassfish.hk2:hk2-locator:jar:2.6.1:compile
[INFO] |  |  +- org.glassfish.hk2.external:aopalliance-repackaged:jar:2.6.1:compile
[INFO] |  |  +- org.glassfish.hk2:hk2-api:jar:2.6.1:compile
[INFO] |  |  \- org.glassfish.hk2:hk2-utils:jar:2.6.1:compile
[INFO] |  \- org.javassist:javassist:jar:3.25.0-GA:compile
[INFO] +- org.glassfish.jersey.containers:jersey-container-servlet:jar:2.35:compile
[INFO] |  +- org.glassfish.jersey.containers:jersey-container-servlet-
core:jar:2.35:compile
[INFO] |  \- org.glassfish.jersey.core:jersey-server:jar:2.35:compile
[INFO] |     \- jakarta.validation:jakarta.validation-api:jar:2.0.2:compile
[INFO] +- com.fasterxml.jackson.core:jackson-databind:jar:2.13.1:compile
[INFO] |  +- com.fasterxml.jackson.core:jackson-annotations:jar:2.13.1:compile
[INFO] |  \- com.fasterxml.jackson.core:jackson-core:jar:2.13.1:compile
[INFO] +- com.fasterxml.jackson.jaxrs:jackson-jaxrs-json-provider:jar:2.13.1:compile
[INFO] |  +- com.fasterxml.jackson.jaxrs:jackson-jaxrs-base:jar:2.13.1:compile
[INFO] |  \- com.fasterxml.jackson.module:jackson-module-jaxb-
annotations:jar:2.13.1:compile
[INFO] |     +- jakarta.xml.bind:jakarta.xml.bind-api:jar:2.3.3:compile
[INFO] |     \- jakarta.activation:jakarta.activation-api:jar:1.2.1:compile
[INFO] +- com.fasterxml.jackson.datatype:jackson-datatype-jsr310:jar:2.13.1:compile
[INFO] +- org.apache.tomcat.embed:tomcat-embed-core:jar:9.0.56:compile
```

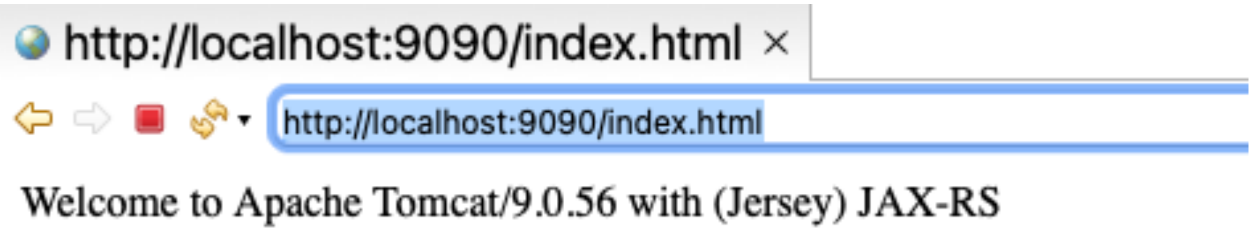
These days, there are many service providers that run Java software on *Cloud-based* servers. The main advantage is that an organization no longer needs to buy/build/install/maintain a bunch of servers (in a dark lonely room in the basement of a building, some places I've seen ... yikes! 😉). Recently, these service providers have been offering servers that are counter-intuitive: they are *smaller* with less CPU, RAM and/or diskspace. Why would they do this? The trade-off of no longer owning servers means that their expenses are now ***monthy***: you essentially rent a server instead of owning a server. Hopefully your business is successful at convincing customers to pay YOU monthly as well. Thus the incentive is to reduce the monthly bill, which leads to smaller, less capable servers for a reduced cost ... this is where Embedded Tomcat comes into play. In my Eclipse IDE I added a JVM argument to my 'launcher' (aka 'Run Configuration') for my `StartEmbeddedTomcat` **`main`** method: `-Xms128M -Xmx128M` which means 'start with 128M of RAM, allow no more than 128M of RAM'. In comparison, the (default) `domain.xml` file for Payara configures 4X as much RAM (512M) and in production it is not uncommon to see

2 or 4G of RAM configured! [Side-note: ***everything*** we are doing with Embedded Tomcat can also be done with the Payara Server]

In Brightspace, you will find the Lab #4 Skeleton .zip-file ‘TomcatEmbedded.zip’ – expand it to some folder and then please import it into Eclipse. This ‘regular’ Java SE application supports a number of REST endpoints:

- `/api/v1/echo`: returns an HTTP 200 ‘Ok’ message with a JSON body of whatever String was `@POST`’d (and a timestamp)
- `/api/v1/helloworld`: returns HTTP 200 ‘Ok’ with a body containing the JSON representation of the `EntityA` class
- `/api/v1/goodbyeworld`: returns HTTP 200 ‘Ok’ with a body containing the JSON representation of the `EntityB` class
- `/api/v1/managed-ref`: returns HTTP 200 ‘Ok’ with more complex JSON body (see below)
- `/api/v1/c2d` and `/api/v1/d2c` : more complex JSON body examples

Test to see if the application started:



http://localhost:9090/index.html ×

http://localhost:9090/index.html

Welcome to Apache Tomcat/9.0.56 with (Jersey) JAX-RS

From the command-line (also can be sent using Postman graphical tool):

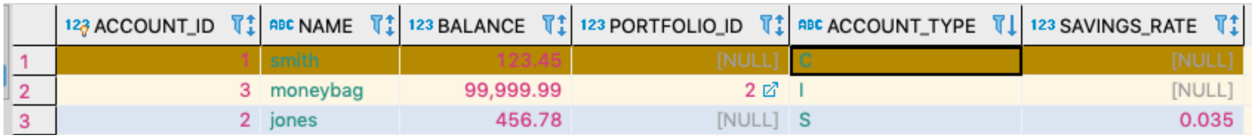
```
prompt> curl http://localhost:9090/api/v1/helloworld
{
  "id" : 2,
  "version" : 1,
  "created" : "2022-01-12T11:17:08.633249",
  "foobar" : null
}

prompt> curl -d 'this message will be echoed back' -H "Content-Type:
text/plain" -X POST http://localhost:9090/api/v1/echo
{
  "date" : "2022-01-12T11:20:13.665086",
  "message" : "this message will be echoed back"
}
```

JSON Mapping

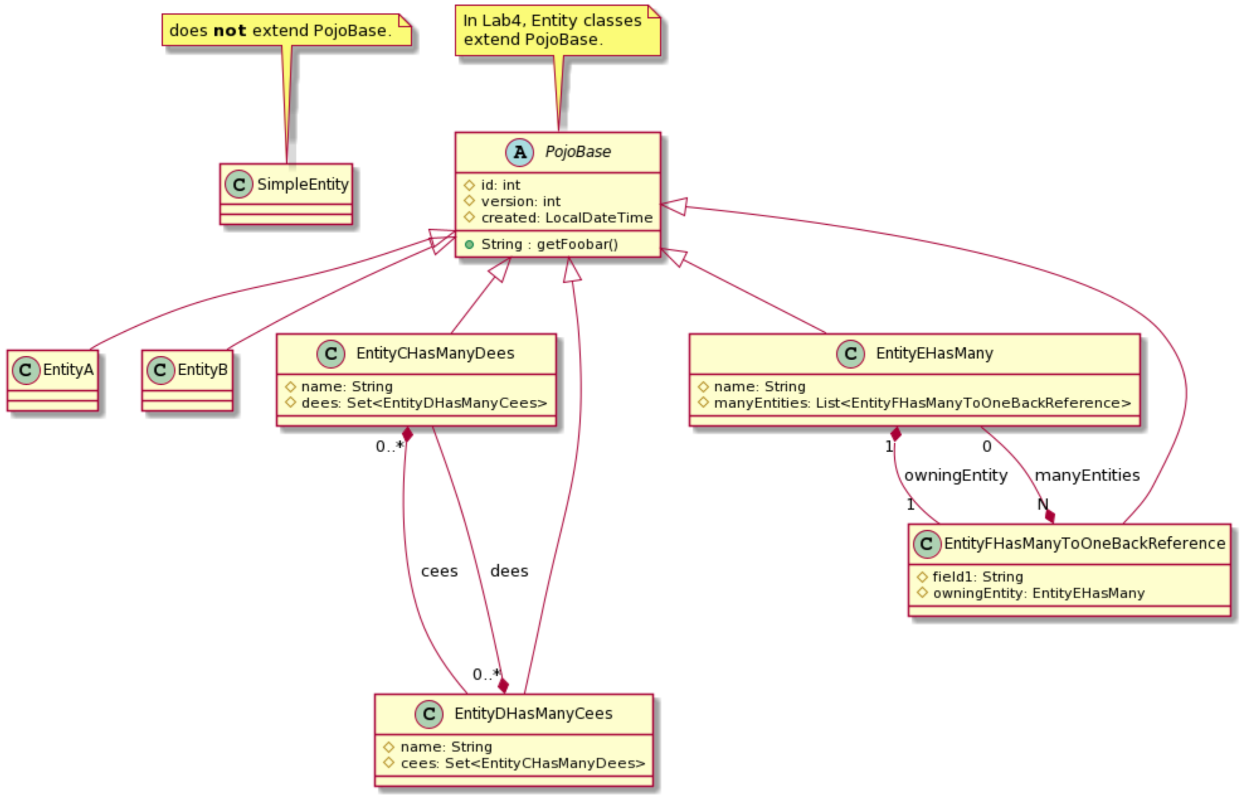
Inheritance

With JPA we need to map between the Java Inheritance hierarchy and the database row using standard JPA annotations `@Inheritance` and `@DiscriminatorColumn`. For example, we can see in the following `ACCOUNTS` table, there is a column `ACCOUNT_TYPE` with codes: ‘C’ which indicates a ChequingAccount, ‘I’ for InterestAccount and ‘S’ for SavingsAccount entities:



	ACCOUNT_ID	NAME	BALANCE	PORTFOLIO_ID	ACCOUNT_TYPE	SAVINGS_RATE
1	1	smith	123.45	[NULL]	C	[NULL]
2	3	moneybag	99,999.99	2	I	[NULL]
3	2	jones	456.78	[NULL]	S	0.035

We can do the same using the Jackson library – while its annotations are not standard, they do accomplish what we need:



If we generated JSON from EntityA and EntityB without any additional Jackson annotations, we cannot tell which JSON message maps to which class:

```
public class EntityA extends PojoBase implements Serializable {
    private static final long serialVersionUID = 1L;

    public EntityA() {
        super();
    }
}
public class EntityB extends PojoBase implements Serializable {
    private static final long serialVersionUID = 1L;

    public EntityB() {
        super();
        //hard-code a message
        setFoobar("bee");
    }
}

    ObjectMapper mapper = new
        ObjectMapper().enable(SerializationFeature.INDENT_OUTPUT);
    mapper.registerModule(new JavaTimeModule());
    mapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
    String entityA_str = mapper.writeValueAsString(new EntityA());
    String entityB_str = mapper.writeValueAsString(new EntityB());
    System.out.println(entityA_str);
    System.out.println(entityB_str);
Console:
{
  "id" : 0,
  "version" : 0,
  "created" : "2022-01-12T14:12:01.043518",
  "foobar" : null
}
{
  "id" : 0,
  "version" : 0,
  "created" : "2022-01-12T14:12:01.043518",
  "foobar" : "bee"
}
```

We use the `@JsonTypeInfo` annotation to cause Jackson to emit an additional JSON field to indicate which Java class the JSON maps to:

```
@JsonTypeInfo (
      use = JsonTypeInfo.Id.NAME,
      include = JsonTypeInfo.As.PROPERTY,
      property = "entity-type")
public class PojoBase implements Serializable {
...
```

Now we can tell which JSON body maps to which Entity:

```
Console:
{
  "entity-type" : "EntityA",
  "id" : 0,
  "version" : 0,
  "created" : "2022-01-12T14:12:01.043518",
  "foobar" : null
}
{
  "entity-type" : "EntityB",
  "id" : 0,
  "version" : 0,
  "created" : "2022-01-12T14:12:01.043518",
  "foobar" : "bee"
}
```

By default, Jackson will use the ‘simple’ Java class name (ie. without the package) – if that is NOT how you wish to encode this information, you can specify a detailed list (minor disadvantage: you must group all the information together in the parent class, unlike JPA’s `@DiscriminatorValue` annotation which is specified on each sub-class):

```
@JsonTypeInfo (
      use = JsonTypeInfo.Id.NAME,
      include = JsonTypeInfo.As.PROPERTY,
      property = "entity-type")
@JsonSubTypes({
      @Type(value = EntityA.class, name = "typeA"),
      @Type(value = EntityB.class, name = "typeB"),
      @Type(value = EntityCHasManyDees.class, name = "typeC"),
      @Type(value = EntityDHasManyCees.class, name = "typeD"),
      @Type(value = EntityEHasMany.class, name = "typeE"),
      @Type(value = EntityFHasManyToOneBackReference.class, name = "typeF")
})
public abstract class PojoBase implements Serializable {
Console:
{
  "entity-type" : "typeA",
  "id" : 0,
  "version" : 0,
  "created" : "2022-01-12T14:12:01.043518",
  "foobar" : null
}
{
  "entity-type" : "typeB",
  "id" : 0,
  "version" : 0,
  "created" : "2022-01-12T14:12:01.043518",
  "foobar" : "bee"
}
```

The code in the Skeleton ‘TomcatEmbedded.zip’ application is missing the `@JsonTypeInfo` and `@JsonSubTypes` annotations: you must add them in order for the JUnit testcases to pass.

A little tidying up ...

We can see above that there is a member field in `PojoBase` called `foobar`; however, we wish to alter how it is displayed in a JSON body. Without changing the name of the field or changing the name of its getter/setter’s, we can use another Jackson Annotation `@JsonProperty`:

```
@JsonProperty("msg")
public String getFoobar() {
      return foobar;
}
public void setFoobar(String foobar) {
      this.foobar = foobar;
}
Console: {
  "entity-type" : "typeB",
  "id" : 0,
  "version" : 0,
  "created" : "2022-01-12T14:12:01.043518",
  "msg" : "bee"
}
```

The code in the Skeleton is missing the `@JsonProperty` annotation: please add it to the `PojoBase` and `SimpleEntity` classes.

Another thing that some folks really do not like is in the EntityA’s JSON message body: it takes over a dozen characters to indicate that a field is **`null`**. Again, Jackson handles this with another annotation `@JsonInclude(JsonInclude.Include.NON_NULL)` (meaning ‘only non-null values appear in JSON body’):

```
@JsonInclude(JsonInclude.Include.NON_NULL)
public abstract class PojoBase implements Serializable {
```

Please add this annotation to **all** the classes in the `c.a.c.l.modelentities` package.

```
Console:
{
  "entity-type" : "typeA",
  "id" : 0,
  "version" : 0,
  "created" : "2022-01-12T14:12:01.043518"
}
```

The last 'tidy-up' is to handle the situation where there is a member field that you do NOT want to appear in the JSON body at all, similar to JPA's `@Transient`. In `SimpleEntity`, there is a member field called `somethingElse` – use the `@JsonIgnore` annotation to make Jackson skip it when serializing the JSON body:

```
@JsonIgnore
public String getSomethingElse() {
      return somethingElse;
}
```

One-to-Many and Many-to-One Relationship

With JPA we need to map the member field for an Entity that has a Collection of 'other' Entities, and for each member of that collection, a member field that holds the 'owner' relationship (JPA annotations `@OneToMany` and `@ManyToOne`)

Jackson has annotations to handle this – if they are NOT used, then an infinite-loop occurs when trying to create the JSON message body:

```
ObjectMapper mapper = new
        ObjectMapper().enable(SerializationFeature.INDENT_OUTPUT);
    mapper.registerModule(new JavaTimeModule());
    mapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
    EntityEHasMany owner = new EntityEHasMany();
    owner.setId(6);
    owner.setVersion(1);
    owner.setName("Bugs Bunny");
    EntityFHasManyToOneBackReference backRef1 = new
        EntityFHasManyToOneBackReference();
    backRef1.setId(8);
    backRef1.setField1("aaa");
    backRef1.setVersion(2);
    backRef1.setOwningEntity(owner);
    owner.getManyEntities().add(backRef1);
    EntityFHasManyToOneBackReference backRef2 = new
        EntityFHasManyToOneBackReference();
    backRef2.setId(11);
    backRef2.setField1("bbb");
    backRef2.setVersion(1);
    backRef2.setOwningEntity(owner);
    owner.getManyEntities().add(backRef2);
    String owner_str = mapper.writeValueAsString(owner);
    System.out.println(owner_str);
Console:
Exception in thread "main"
    com.fasterxml.jackson.databind.JsonMappingException: Infinite recursion
    (StackOverflowError) (through reference chain:
    c.a.c.lab.modelentities.EntityEHasMany["manyEntities"] -
        >java.util.ArrayList[0]-
        >c.a.c.lab.modelentities.
            EntityFHasManyToOneBackReference["owningEntity"]
[Note: the details of the JsonMappingException continue for many lines]
```

Once the Jackson annotations `@JsonManagedReference` and `@JsonBackReference` are in place, the output now makes sense:

```
public class EntityEHasMany extends PojoBase implements Serializable {
    /** explicit set serialVersionUID */
    private static final long serialVersionUID = 1L;

    protected List<EntityFHasManyToOneBackReference> manyEntities = new
ArrayList<>();

...

    @JsonManagedReference
    public List<EntityFHasManyToOneBackReference> getManyEntities() {
        return manyEntities;
    }
public class EntityFHasManyToOneBackReference extends PojoBase implements
Serializable {
    /** explicit set serialVersionUID */
    private static final long serialVersionUID = 1L;

    protected EntityEHasMany owningEntity;

...

    @JsonBackReference
    public EntityEHasMany getOwningEntity() {
        return owningEntity;
    }
Console:
{
    "entity-type": "typeE",
    "id": 6,
    "version": 1,
    "created": "2022-01-13T10:02:24.444965",
    "name": "Bugs Bunny",
    "manyEntities": [
        {
            "entity-type": "typeF",
            "id": 8,
            "version": 2,
            "created": "2022-01-13T10:02:24.446303",
            "field1": "aaa"
        },
        {
            "entity-type": "typeF",
            "id": 11,
            "version": 1,
            "created": "2022-01-13T10:02:24.446326",
            "field1": "bbb"
        }
    ]
}
```

The code in the Skeleton is missing the `@JsonManagedReference` and `@JsonBackReference` annotations: please add them.

Many-to-Many Relationship

The Many-to-Many relationship is the most complex to represent – for example, a Database needs three (3) tables to represent it. In Java code, it is deceptively simple: an Entity has a collection of other entities, similar to the OneToMany scenario. However, for each member of that collection, there is not a single ‘owner’ – each can be in multiple collections. A typical example would be `StudentEntity` and `CourseEntity` – for a particular course, we can ‘see’ the collection of registered students; however, those students can be registered in many courses:

```
public class CourseEntity {
    Set<StudentEntity> registeredStudents;
}
public class StudentEntity {
    Set<CourseEntity> registeredCourses;
}
```

Jackson does not have a specific annotation to handle this scenario. However, fortunately there is an annotation that tells the system – ‘please follow the instructions of this custom Serializer’ – so we can ‘MacGyver’ a solution (https://youtu.be/09UlB17cgKw?t=117, no paperclips were harmed in preparing Lab 4 😉):

```
public class EntityCHasManyDees extends PojoBase implements Serializable {
    /** explicit set serialVersionUID */
    private static final long serialVersionUID = 1L;
...

    @JsonSerialize(using = SetOfEntityDeesSerializer.class)
    public Set<EntityDHasManyCees> getDees() {
        return dees;
    }
```

Just as with the OneToMany scenario, if we do not Serialize the set of ‘D’ entities properly, an infinite-loop will occur. Thinking about the JSON body, once we have serialized the information for a particular instance of ‘C’, the ‘D’s will next be serialized … but for each ‘D’, we really do not need to ‘see’ its collection of ‘C’s. Thus, the strategy in `SetOfEntityDeesSerializer`, is to ‘hollow-out’ the set:

```
public class SetOfEntityDeesSerializer extends
    StdSerializer<Set<EntityDHasManyCees>> {
    @Override
    public void serialize(Set<EntityDHasManyCees> originalDees,
        JsonGenerator gen, SerializerProvider provider) throws
        IOException {
        Set<EntityDHasManyCees> hollowDees = new HashSet<>();
        for (EntityDHasManyCees originalDee : originalDees) {
            // create a 'hollow' copy of the original
            // EntityDHasManyCees
            EntityDHasManyCees hollowDee = new EntityDHasManyCees();
            hollowDee.setCreated(originalDee.getCreated());
            hollowDee.setId(originalDee.getId());
            hollowDee.setVersion(originalDee.getVersion());
            hollowDee.setName(originalDee.getName());
            hollowDee.setFoobar(originalDee.getFoobar());
            hollowDee.setCees(null); // this prevents infinte-loop
            hollowDees.add(hollowDee);
        }
        gen.writeObject(hollowDees);
    }
}
```

The code in the Skeleton contain the custom Serializers; however, it must be added to `EntityCHasManyDees` and `EntityDHasManyCees`.

```
prompt> curl http://localhost:9090/api/v1/c2d
{
  "entity-type" : "EntityCHasManyDees",
  "id" : 5,
  "version" : 0,
  "created" : "2022-01-13T10:34:34.727279",
  "name" : "qqq",
  "dees" : [ {
    "id" : 7,
    "version" : 0,
    "created" : "2022-01-13T10:34:34.728036",
    "name" : "eee"
  } ]
}

prompt> curl http://localhost:9090/api/v1/d2c
{
  "entity-type" : "EntityDHasManyCees",
  "id" : 9,
  "version" : 0,
  "created" : "2022-01-13T10:34:34.728055",
  "name" : "ttt",
  "cees" : [ {
    "entity-type" : "EntityCHasManyDees",
    "id" : 6,
    "version" : 0,
    "created" : "2022-01-13T10:34:34.72801",
    "name" : "www",
    "dees" : [ {
      "id" : 9,
      "version" : 0,
      "created" : "2022-01-13T10:34:34.728055",
      "name" : "ttt"
    }, {
      "id" : 8,
      "version" : 0,
      "created" : "2022-01-13T10:34:34.728047",
      "name" : "rrr"
    } ]
  }, {
    "entity-type" : "EntityCHasManyDees",
    "id" : 5,
    "version" : 0,
    "created" : "2022-01-13T10:34:34.727279",
    "name" : "qqq",
    "dees" : [ {
      "id" : 7,
      "version" : 0,
      "created" : "2022-01-13T10:34:34.728036",
      "name" : "eee"
    } ]
  } ]
}
```

- end -